Object Communication and Events

Behavioral Design Patterns

SoftUni Team

Technical Trainers

Software University

http://softuni.bg

Table of Contents

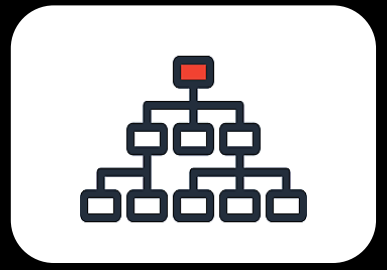
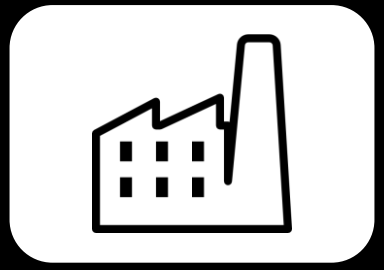
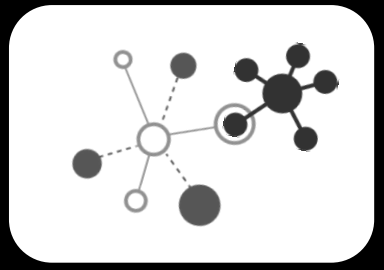
Questions

SoftUni Foundation

sli.do

#JavaFundamentals

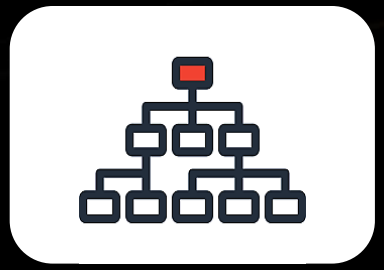
SoftUni Foundation

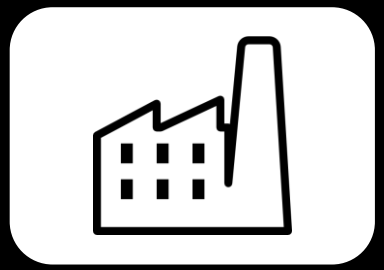




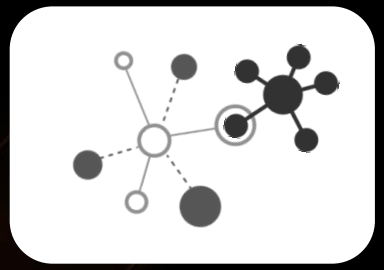
Design Pattern

Common Solutions to Common Problems

Design Patterns

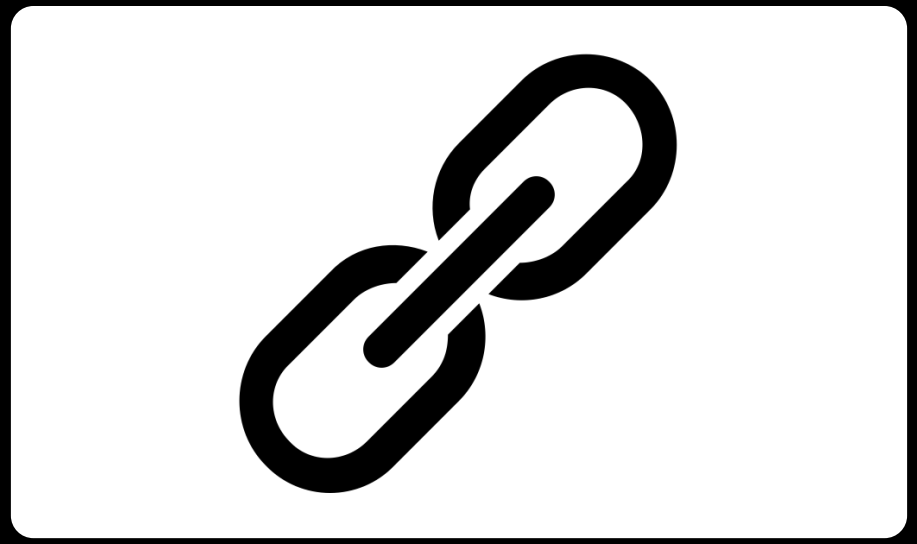






- **Structural**
  - All about Class and Object **composition**
- **Creational**
  - All about Object **creation** mechanisms
- **Behavioral**
  - All about Object **communication**

SoftUni Foundation



Chain of Responsibility

Decoupling Requests

Chain of Responsibility

SoftUni Foundation

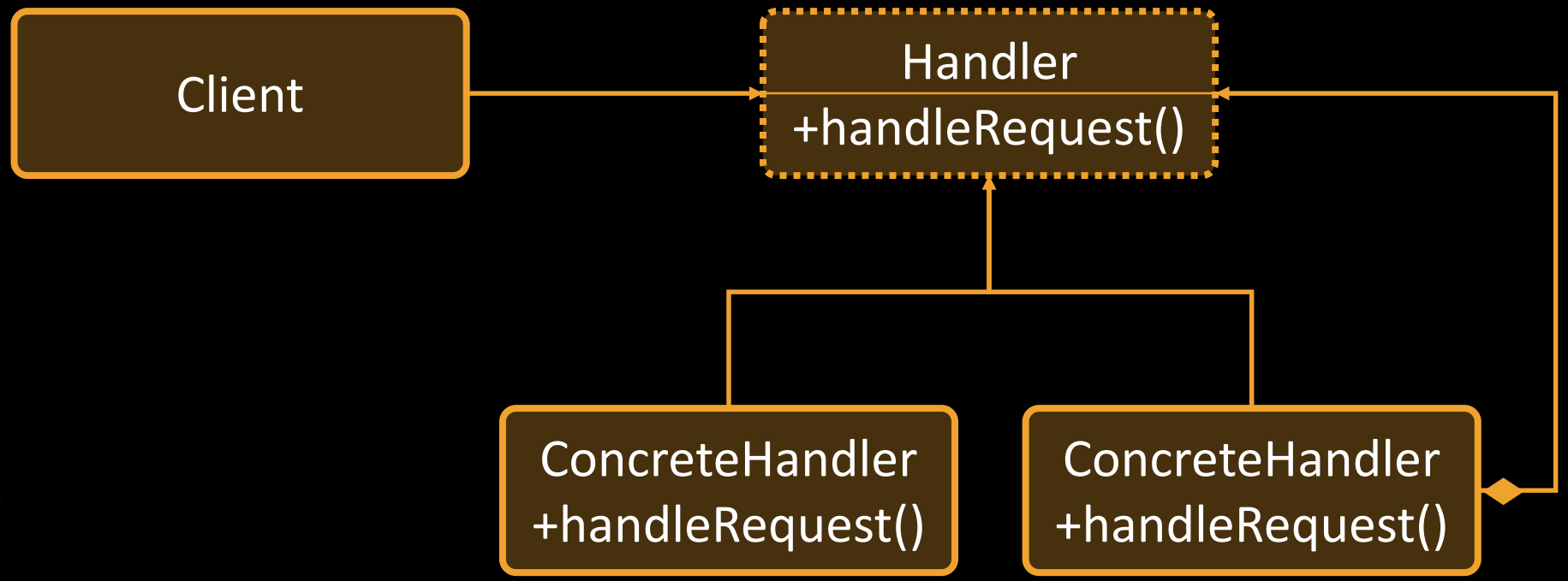
- **Decouples** sender and receiver
- **Chains multiple receivers** that can handle a request
- Supports **undoable requests**

```
Logger requestLogger = new Logger();
Logger messageLogger = new Logger();
requestLogger.next(messageLogger);

requestLogger.log("…");
```

Chain of Responsibility – UML

- Handler, ConcreteHandler



Problem: Logger

SoftUni Foundation

- Create a **Chain of Responsibility** Logger and provide:
- `enum LogType` (ATTACK, MAGIC, TARGET, ERROR, EVENT)
- `interface Handler`
  - `void handle(LogType, String)`
  - `void setSuccessor(Handler)`
- Concrete loggers that log messages to console:
  - `CombatLogger`, `EventLogger`
  - Log in format (`"TYPE: message"`)

Solution: Logger

SoftUni Foundation

```
public interface Handler {
  void handle(RequestType type, String message);
  void setSuccessor(Handler handler);
}
```

```
public enum RequestType {
  ATTACK, MAGIC, TARGET, ERROR, EVENT
}
```

Solution: Logger (2)

SoftUni Foundation

```
public abstract class Logger implements Handler {
  private Handler successor;
  public void setSuccessor(Handler successor) {
    this.successor = successor;
  }
  protected void passToSuccessor(…) {
    if (this.successor != null) {
      this.successor.handle(type, message);
    }
  }
  public abstract void handle(…);
}
```

Solution: Logger (3)

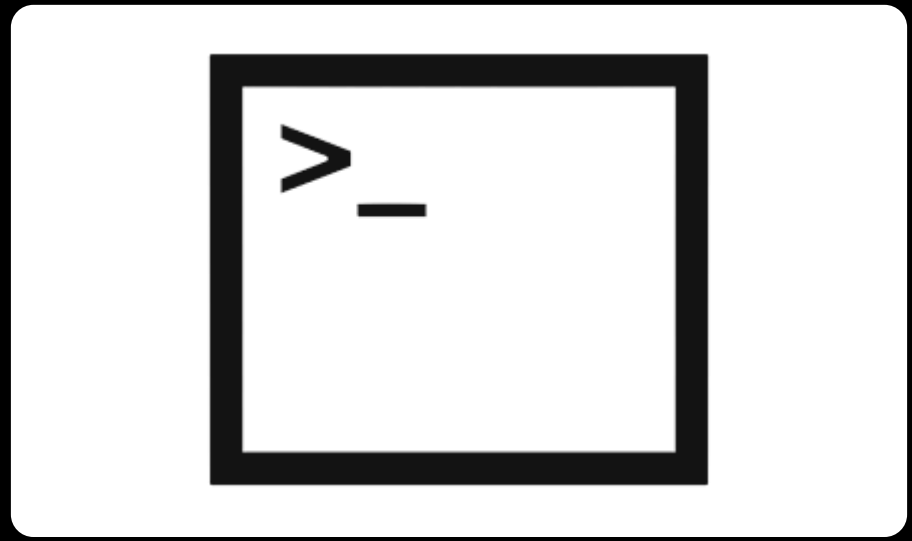
SoftUni Foundation

```
public class CombatLogger extends Logger {

  @Override
  public void handle(…) {
    if (type == RequestType.ATTACK) {
      System.out.println(
              type.name() + ": " + message);
    }

    super.passToSuccessor(type, message);
  }
}
```

SoftUni Foundation



Command Pattern

Encapsulate Requests as an Objects

Command Design Pattern

SoftUni Foundation

- **Callbacks** are now **Object Oriented**
- **Extending** behavior is more **flexible**
- **Decouples** the **Invoker** from the **Receiver**

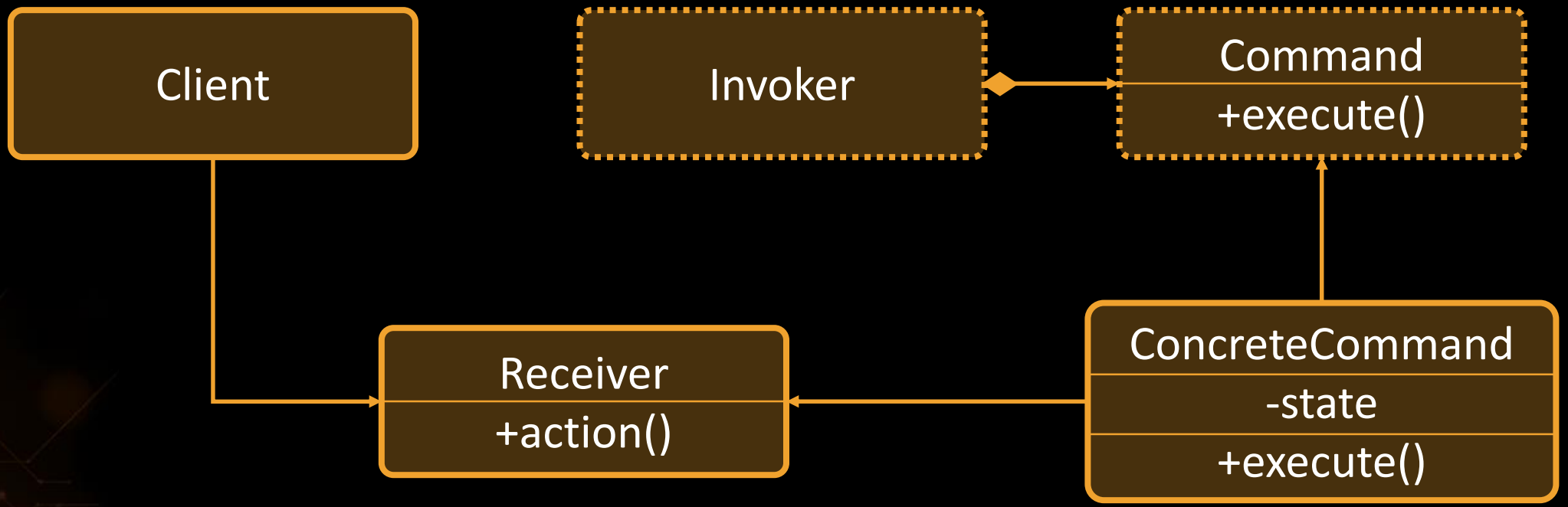
Callbacks == Functions/Methods

```
Executor executor = new CommandExecutor();
Receiver receiver = new CommandReceiver();
Request request = new Request(receiver);

executor.execute(request);
```

Command – UML

- Invoker, Receiver
- Command, ConcreteCommand



Problem: Command

SoftUni Foundation

- Create a **Command Pattern** Executor and provide:
- `interface Command`
  - `void execute()`
- `interface Executor`
  - `void executeCommand(Command command)`
- Concrete Executor named `CommandExecutor`
- Concrete Commands
  - `TargetCommand(Attacker, Target)`
  - `AttackCommand(Attacker)`

Solution: Command Executor

SoftUni Foundation

```
public interface Command {
    void execute();
}
```

```
public interface Executor {
    void executeCommand(Command command);
}
```

```
class CommandExecutor implements Executor {
  public void executeCommand(Command command) {
    command.execute();
  }
}
```

Solution: Command Executor (2)

SoftUni Foundation

```
public class AttackCommand implements Command {

  private Attacker attacker;

  public AttackCommand(Attacker attacker) {
    this.attacker = attacker;
  }

  public void execute() {
    this.attacker.attack();
  }
}
```

SoftUni Foundation

Chain of Responsibility, Command

Live Exercises in Class (Lab)

SoftUni Foundation

Mediator

Handling Groups of Colleagues

Mediator Design Pattern

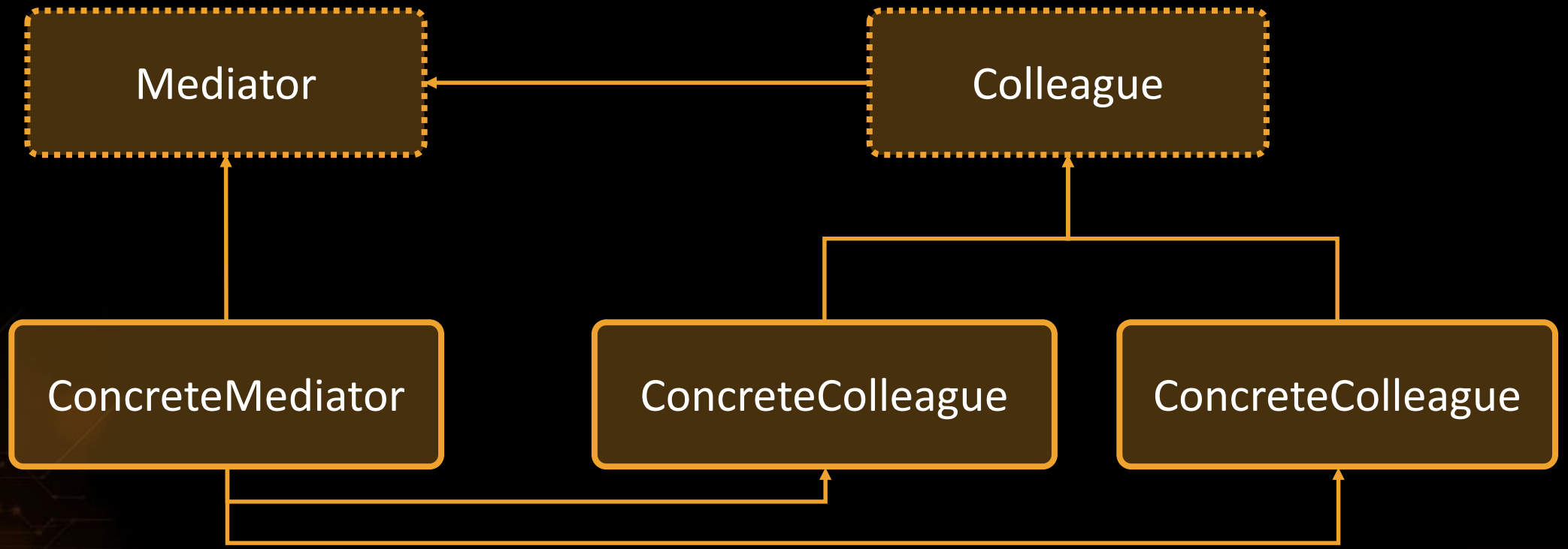
SoftUni Foundation

- **Encapsulate** how a set of **objects interact**
- **Colleagues** are **decoupled** to one another

```
Mediator mediator = new GroupController();
mediator.addColleague(new Colleague());
mediator.addColleague(new Colleague());
mediator.moveAll();
mediator.updateAll();
```

Mediator – UML

- Mediator, Colleague
- ConcreteMediator, ConcreteColleague



Problem: Group

- Create a Mediator and provide:
- `interface AttackGroup`
  - `void addMember(Attacker)`
  - `void groupTarget(Target)`
  - `void groupAttack()`
- Concrete class `Group` that implements `AttackGroup`
- Concrete Commands:
  - `GroupTargetCommand(AttackGroup, Target)`
  - `GroupAttackCommand(AttackGroup)`

Solution: Group

SoftUni Foundation

```
public interface AttackGroup {
    void addMember(Attacker attacker);
    void groupTarget(Target target);
    void groupAttack();
}
```

Solution: Group (2)

SoftUni Foundation

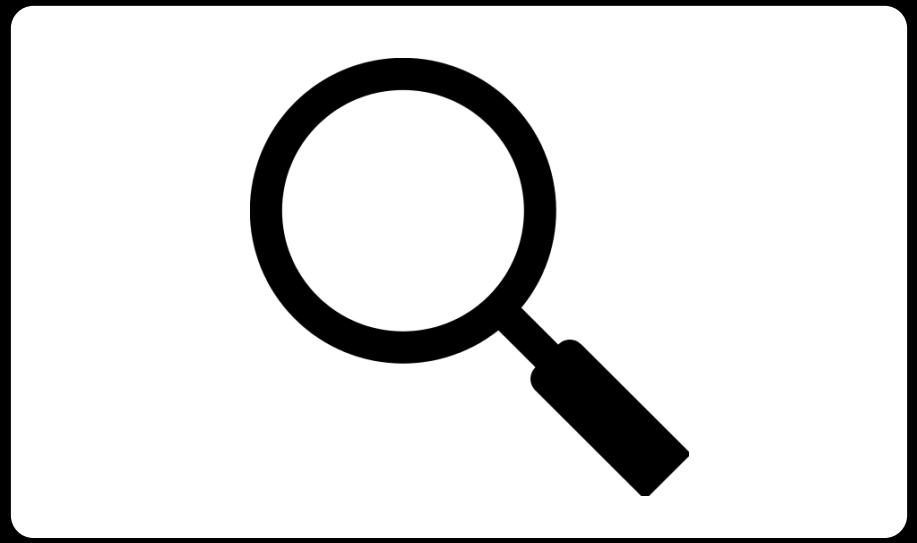
```
public class Group implements AttackGroup {
  private List<Attacker> attackers;
  public Group() {
    this.attackers = new ArrayList<>();
  }
  public void addMember(Attacker attacker) { … }
  public void groupTarget(Target target) { … }
  public void groupAttack() { … }
}
```

Solution: Group (3)

SoftUni Foundation

```
public class GroupTargetCommand implements Command {
  private AttackGroup group;
  private Target target;
  public GroupTargetCommand(AttackGroup group, Target target) {
    this.group = group;
    this.target = target;
  }
  public void execute() {
    this.group.groupTarget(this.target);
  } }
```

SoftUni Foundation



Observer

Handle Events

Observer Design Pattern

SoftUni Foundation

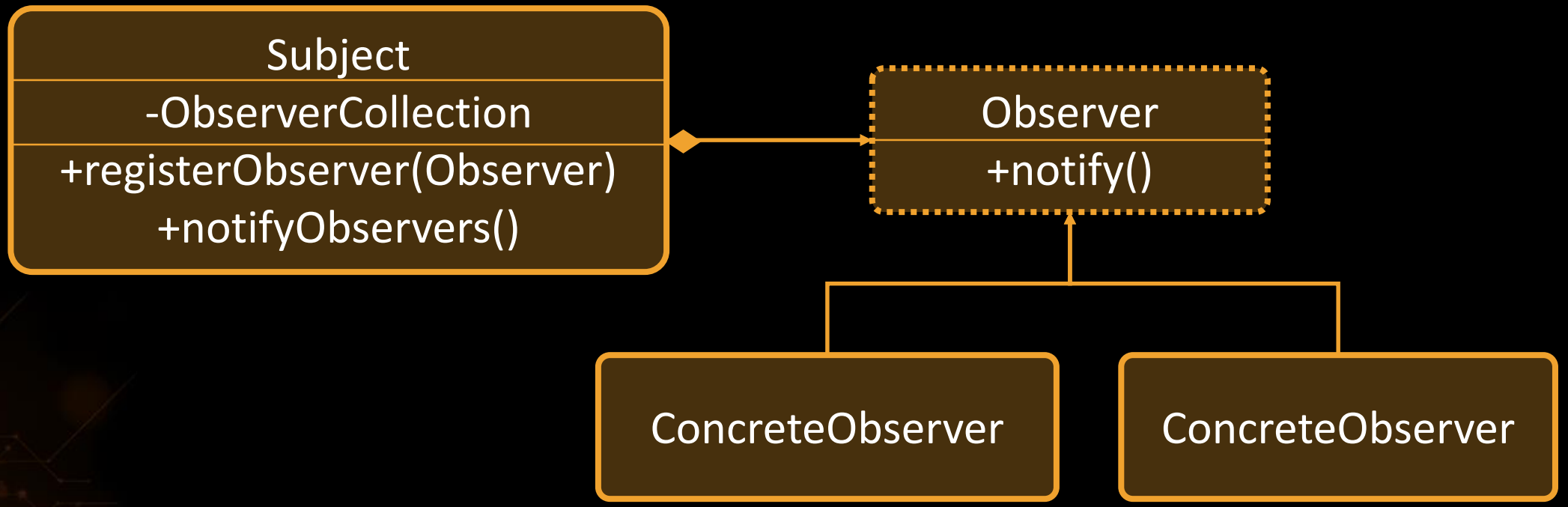
- Define a **one-to-many** relationship
- **Update observers** once an event in the subject occurs

```
Subject subject = new Subject();
subject.addObserver(new Observer());
mediator.addObserver(new Observer());

// observers are notified after a state change
```

Observer – UML

- Subject, Observer
- ConcreteObserver



Problem: Observer

- Implement the following:
- interface `Subject`
  - `void register(Observer)`
  - `void unregister(Observer)`
  - `void notifyObservers()`
- interface `Observer`
  - `update(int)`
- If a **Target** dies, it should **send reward** to all of its **Observers**

Solution: Observer

SoftUni Foundation

```
public interface Subject {
    void register(Observer observer);
    void unregister(Observer observer);
    void notifyObservers();
}
```

```
public interface Target extends Subject {
    …
}
```

* This is **violation** of **ISP**, find a better solution

Solution: Observer

SoftUni Foundation

```
public interface Observer {
    void update(int val);
}
```

```
… class Hero implements Attacker, Observer {


  // implementation
}
```

Solution: Observer

SoftUni Foundation

```
public void register(Observer observer) {
  this.observers.add(observer);
}
public void unregister(Observer observer) {
  this.observers.remove(observer);
}
                          //Continues on next slide
```

Add methods to Dragon implementation

Solution: Observer(2)

SoftUni Foundation

```
//…
public void notifyObservers() {
  for (Observer observer : observers) {
    observer.update(this.reward);
  }
}
```

Summary

- Design Patterns, are **common solutions** to **common problems**
- To learn more about **object communication**:
  - Practice **behavioural design patterns**
  - **Pick a pattern** and think of a **specific problem** where you can use it
  - **Code** the solution that you've come up with
- The same applies for object creation (**Creational patterns**) and class structure (**Structural patterns**)

Object Communication and Events

SoftUni Foundation

Questions?

https://softuni.bg/courses/

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg
- Software University Foundation
  - http://softuni.foundation/
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

Software University

SoftUni Foundation

FORUM

License

SoftUni Foundation

- This course (slides, examples, demos, videos, homework, etc.) is licensed under the "Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International" license

- Attribution: this work may contain portions from
  - "Fundamentals of Computer Programming with Java" book by Svetlin Nakov & Co. under CC-BY-SA license
  - "C# Part I" course by Telerik Academy under CC-BY-NC-SA license
  - "C# Part II" course by Telerik Academy under CC-BY-NC-SA license